Automatic Generation of Directive-Based Parallel Programs for Shared Memory Parallel Systems

H. Jin, J. Yan and M. Frumkin
NAS System Division, NASA Ames Research Center, Moffett Field, CA 94035-1000
{hjin,yan,frumkin}@nas.nasa.gov

Abstract

The shared-memory programming model is a very effective way to achieve parallelism on shared memory parallel computers. As great progress was made in hardware and software technologies, performance of parallel programs with compiler directives has demonstrated large improvement. The introduction of OpenMP directives, the industrial standard for shared-memory programming, has minimized the issue of portability. Due to its ease of programming and its good performance, the technique has become very popular. In this study, we have extended CAPTools, a computer-aided parallelization toolkit, to automatically generate directive-based, OpenMP, parallel programs. We outline techniques used in the implementation of the tool and present test results on the NAS parallel benchmarks and ARC3D, a CFD application. This work demonstrates the great potential of using computer-aided tools to quickly port parallel programs and also achieve good performance.

Keywords: shared memory programming, OpenMP directives, computer-aided tools, automated parallel code generation.

1 Introduction

Over the past decade, high performance computers based on commodity microprocessors have been introduced in rapid succession from many vendors. These include Intel's IPSC/860, Delta and Paragon, TMC's CM-2 and CM-5, IBM's SP1 and SP2, Cray's T3D and T3E, SGI Origin 2000, PowerChallenge Onyx2, and Sun's Enterprise and HPC2 servers. These systems could be classified into two major categories: distributed memory (DMP) and shared memory parallel (SMP) systems. While both categories consist of physically distributed memories, the hardware support for shared memory machines presents a globally shared address space. While all these architectures support some form of message passing (e.g. MPI, the de facto standard today), the SMP's further support program annotation standards, or directives, that allow the user to supply information to the compiler to assist in the parallelization of the code. The notion of the use of directives is not new, they have been used in many cases to allow machine-specific optimization. Because the directives are inserted as comments, the program will maintain portability across

machines, while it will run exceptionally fast on the machine for which these directives are written.

Currently, there are two widely accepted standards for annotating programs for parallel executions: High Performance Fortran (HPF) and OpenMP. HPF [6] provides a data parallel model of computation for DMP systems. OpenMP [15], on the other hand, is designed to offer a portable solution for parallel programming on SMP systems with compiler directives. OpenMP is, in a sense, "orthogonal" to the HPF type of parallelization because computation is distributed inside a loop based on the index range regardless of data location. There are many advantages in using compiler directives as opposed to writing message passing programs:

- no need for explicit data partitioning,
- scalability by taking advantage of hardware cache coherence,
- portability via standardization activities (such as OpenMP), and
- simple to program, with incremental approach to code development.

Perhaps the main disadvantage of programming with directives is that inserted directives may not necessarily enhance performance. In the worst cases, it can create erroneous results when used incorrectly (writing message passing codes is even more error-prone). While vendors have provided tools to perform error-checking and profiling [13], automation in directive insertion is very limited and often failed on large programs, primarily due to the lack of a thorough enough data dependence analysis. To overcome the deficiency, we have developed a toolkit, CAPO, to automatically insert OpenMP directives in Fortran programs and apply a degree of optimization. CAPO is aimed at taking advantage of detailed interprocedural data dependence analysis provided by CAPTools (Computer-Aided Parallelization Tools) [7], developed by the University of Greenwich, to reduce potential errors made by users and, with small amount of help from user, achieve performance close to that obtained when directives are inserted by hand.

In the following, we first outline the process of programming using a shared memory model. Then, in Section 2 we give an overview of CAPTools and discuss its extension CAPO for generating OpenMP programs. The implementation of CAPO is discussed in Section 3. Results of two test cases with CAPO are presented in Section 4 and conclusions are given in the last Section.

1.1 Shared-Memory Programming Model

The shared memory model is a natural extension of the sequential programming model because of a globally accessible address space. Users may ignore the interconnection details of parallel machines and exploit parallelism with a minimum of difficulty. Insertion of compiler directives into a serial program to generate a parallel program eases the job of porting applications to high performance parallel computers.

An SMP program follows a simple *fork-and-join* execution model. The fork-and-join program initializes as a single light weight process, called the *master thread*. The master thread executes sequentially until the first parallel construct is encountered. At that point, the master thread creates a team of threads, including itself as a member of the team, to concurrently execute the statements in the parallel construct. When a worksharing construct such as a parallel do is encountered, the workload is distributed among the members of the team. Upon completion of the parallel construct, the threads in the team synchronize and only the master thread continues execution. The fork-join process can be repeated many times in the course of program execution.

1.2 OpenMP Directives

OpenMP [15] was designed to facilitate portable implementation of shared memory parallel programs. It includes a set of compiler directives and callable runtime library routines that extend Fortran, C and C++ to support shared memory parallelism. It promises an incremental path for parallelizing sequential software, as well as targeting at scalability and performance for any complete rewrites or new construction of applications.

OpenMP follows the fork-and-join execution model. A parallel region is defined by the "PARALLEL" and " END PARALLEL" pair. The "PARALLEL DO" or "DO" directive is an example of a worksharing construct, which distributes the workload of a DO loop among the members of the current thread team. An implied synchronization occurs at the end of the DO loop unless an "END DO NOWAIT" is specified. Data sharing of variables is specified at the start of parallel or worksharing constructs using the SHARED and PRIVATE clauses. In addition, reduction operations (such as summation) can be specified by the "REDUCTION" clause.

Beyond the inclusion of parallel constructs to distribute work to multiple threads, OpenMP introduces a powerful concept of *orphan directives* that greatly simplifies the task of implementing coarse grain parallel algorithms. Orphan directives are directives outside the lexical extent of a parallel region. This allows the user to specify control or synchronization from anywhere inside the parallel region, not just from the lexically contained region.

2 Computer-Aided Parallelization

The goal of developing computer-aided tools to help parallelize applications is to let the tools do as much as possible and minimize the amount of tedious and error-prone work performed by the user. The key to automatic detection of parallelism in a program and, thus parallelization, is to obtain accurate data dependences in the program. Of course, we have to realize that there are always cases in which certain conditions could prevent tools from detecting possible parallelization, thus, an interactive user environment is also important. Our goal is to have computer-aided tools automate 90% of the work and identify the other 10% that user should focus on. CAPTools provides such an environment, so we will briefly describe the concepts of CAPTools and then describe our extensions to the toolkit for generating directive-based parallel

codes. In order to facilitate later discussions, it is useful to first mention a few commonly-known concepts.

2.1 Data Dependences and Dependence Analysis

There are four basic types of data dependences in a program source. For two statements, S1 preceding S2, in a program, the execution order of S1 and S2 can not be changed if any one of the following conditions exists:

1) *True* dependence – data is written in S1 and read in S2,
2) *Anti* dependence – data is read in S1 and written in S2,
3) *Output* dependence – data is written in S1 and written again in S2,
4) *Control* dependence – S2 is executed only if S1is tested true.

For a loop, dependences can further be marked as loop-carried or loop-independent, based on the relationship with loop iteration. For instance, the following loop has a loop-carried true dependence of variable A, which prevents the loop from being distributed (or executed in parallel):

```
DO I=1, N
  A(I) = A(I) + A(I-1)
END DO
```

Dependence analysis is one of the fundamental areas in compiler technologies and has been researched for many years. Dependence analysis employs a set of tests in an attempt to disprove the existence of a dependence. The Greatest Common Divisor test [1] and Banerjee's inequality test [5] are the two commonly used algorithms. Since dependence analysis is not the focus of this work, we refer interested readers to the relevant literature ([8] and therein).

2.2 CAPTools

The Computer-Aided Parallelization Tools (CAPTools) is a software toolkit that automates the generation of message-passing parallel code. The parallelization environment of CAPTools is outlined in Figure 1 (for details, see reference [7]). CAPTools accepts FORTRAN-77 serial code as input, performs extensive dependence analysis, and uses domain decomposition to exploit parallelism. The tool employs sophisticated algorithms to calculate execution control masks and minimize communication. The generated parallel codes contain portable interface to message passing standards, such as MPI and PVM, through a low-overhead library.

There are two important strengths that make the tool stands out. Firstly, an extensive set of extensions [8] to the conventional dependence analysis techniques has allowed CAPTools to obtain much more accurate dependence information and, thus, produce more efficient parallel code. Secondly, the tool contains a set of browsers that allow user to inspect and assist parallelization at different stages.

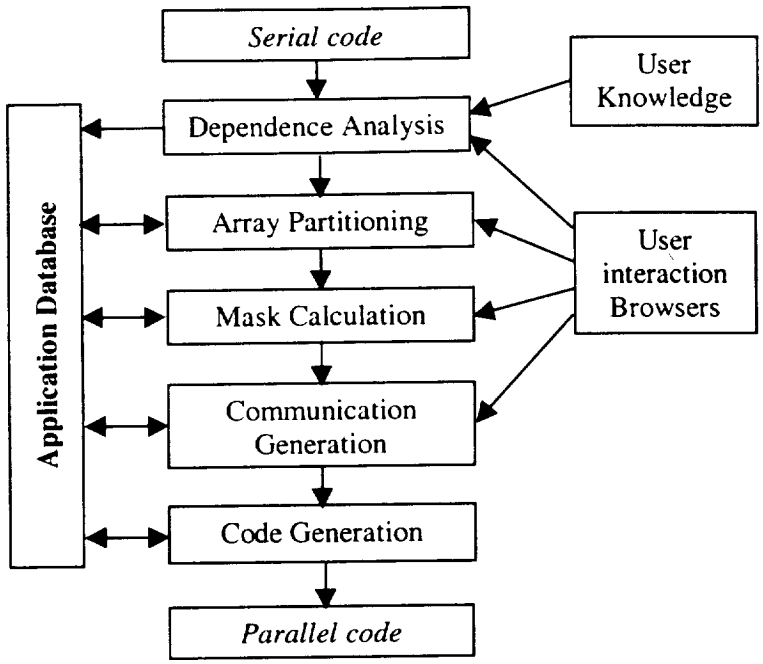

Figure 1: Schematic flow chart of the CAPTools parallelization environment for message passing programs.

2.3 Generating Directive-Based Parallel Programs

To generate directive-based parallel programs for shared memory systems, we developed a CAPTools-based Automatic Parallelizer with OpenMP, CAPO. The tool takes advantage of the accurate data dependence analysis provided by CAPTools and complements CAPTools' functionality. CAPO exploits parallelism at the loop level where a parallel loop is the basic element for parallelization. A parallel loop is a loop without data dependences in loop iterations, thus, the loop can be distributed among multiple threads. On the other hand, if two code blocks can be proved having no data dependences, the two code blocks can be executed concurrently. However, parallelism in this case is rare and is not considered in CAPO.

Figure 2 illustrates the schematic structure of CAPO and Section 3 discusses the detailed implementation of CAPO. A serial Fortran 77 code is loaded into CAPO and a data dependence analysis is performed. User knowledge can be added to assist this process for more accurate results. The process of generating directive-based parallel code can be summarized in the following three main stages.

1) *Identifying parallel loops and parallel regions.* It starts with the loop-level analysis using the data dependence information. Loops are classified as parallel (including reduction), serial or potential pipeline. Distributed loops (i.e. work-sharing directives will be added for parallel execution) are identified by traversing the call graph in a top-down approach and only outer-most

parallel loops are checked. Parallel regions are then formed around the distributed loops. Attempt is also made to create parallel pipelines. Details are given in Sections 3.1-3.3.

2) *Optimizing loops and regions.* This stage is mainly for reducing overhead caused by fork-and-join and synchronization. A parallel region is first expanded as large as possible. Regions are then merged together if there is no violation of data usage in doing so. Region optimization is currently limited to within a subroutine. The synchronization between loops can be removed if the loops can be executed asynchronously. Details are given in Sections 3.2 and 3.4.

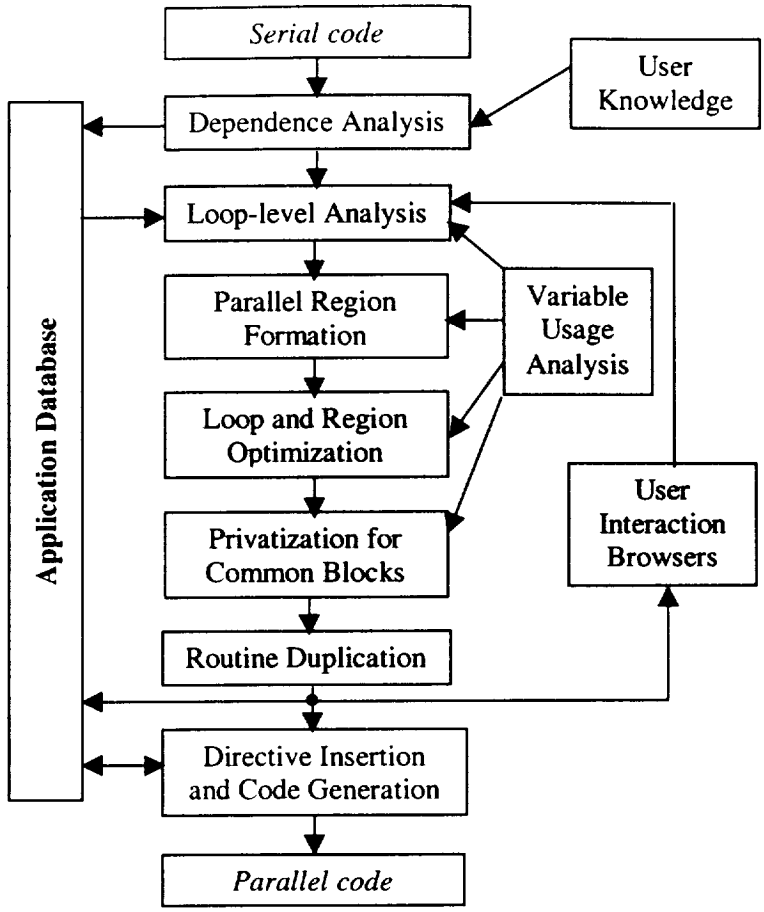

Figure 2: Schematic flow chart of the CAPO architecture.

3) *Transforming codes and inserting directives.* Variables in common blocks are analyzed for their usage in all parallel regions in order to identify threadprivate common blocks. If a private variable is used in a non-threadprivate common block, the variable is treated specially. A routine needs to be duplicated if it is called both inside and outside a distributed loop and contains distributed loops themselves. Details are given in Sections 3.5-3.7. Directives are lastly added to parallel regions and distributed loops by traversing the call graph with variables properly listed.

Additional analysis is performed at several points to identify how variables are used (e.g. private, shared, reduction, etc.) in a specific loop or region. Such an analysis is required for classification of loop types, construction of parallel regions, treatment of private variables in common blocks, and insertion of directives.

Intermediate results can be stored into or retrieved from a database. User assistance to the parallelization process is possible through a browser implemented in CAPO (Directives Browser) and other browsers provided in CAPTools. The Directives Browser is designed to provide more interactive information on reasons why loops are parallel or serial, distributed or

not distributed. User can concentrate on areas where potential improvements could be made, for example, by removing false dependences. It is part of the iterative process of parallelization.

3 Implementation

In the following subsections, we will give some implementation details of CAPO organized according to the components described in Section 2.3.

3.1 Loop-level Analysis

In the loop-level analysis, the data dependence information is used to classify loops in each routine. Loop types include *parallel*, *reduction*, *serial*, and *pipeline*. A parallel loop is a loop with no loop-carried true data dependence and no exiting statements that jump outside the loop (e.g. `RETURN`). Loops with I/O (e.g. `READ`, `WRITE`) statements are excluded from consideration at this point. If the parallel execution of a loop would cause a memory access conflict for some variable and the variable could not be privatized, the loop can only be executed serially.

A reduction loop is similar to parallel loop except that the loop carries true dependences caused by well-defined reduction operations, such as "+", "-", "`min`", "`max`", etc. The reduction sum is an example:

```
SUM = 0
DO I=1, N
  SUM = SUM + A(I)
END DO
```

A special class of loops, called *pipeline loop*, has loop-carried true dependencies and determinable dependence vectors. Such a loop can potentially be used to form parallel pipelining with an outside loop nesting. Compiler techniques for finding pipeline parallelism through affine transforms are discussed in [14]. The pipeline parallelism can be implemented in OpenMP directives with point-to-point synchronization. This is discussed in Section 3.3.

A serial loop is a loop that can not be run in parallel due to loop-carried data dependences, I/O or exiting statements. However, a serial loop can be considered for the formation of a parallel pipeline.

3.2 Setup of Parallel Region

In order to achieve good performance, it is not enough to simply stay with parallel loops at a finer grained level. In the context of OpenMP, it is possible to express coarser-grained parallelism with parallel regions. Our next task is to use the loop-level information to define these parallel regions.

There are several steps to construct parallel regions:

1) Identify distributed loops by traversing the call graph in a top-down approach. Distributed loops can include parallel loops, reduction loops and pipeline loops in the outer-most loop level. These loops are further checked for granularity.

2) Find a distributed loop in a routine, starting from the leaf-node routines in the call graph and working towards the main routine.

3) Move up as much as possible in the routine to the top-most loop nest that contains no I/O and exiting statements and has not been included in a parallel region. If any variable usage would cause memory access conflict in the potential parallel region and the variable can not be privatized, back down one loop nest. A parallel region is then constructed around the distributed loop.

4) Include in the region any preceded code blocks that satisfy the memory conflict test and are not yet included in other parallel regions. Orphaned directives will be used in routines that are called inside a parallel region but outside a distributed loop.

5) Join two neighboring regions to form a larger parallel region if possible. A reduction loop is in a parallel region by itself.

6) If the top loop in the region is a pipeline loop and the loop is also in the top-level loop nesting of its routine, then the loop is further checked for loop nest in the immediate parent routine. The loop can be used to form an "upper" level pipeline region only if all the following tests are true: a) such a parent loop nest exists, b) each routine called inside the parent loop contains only a single parallel region, and c) except for pipeline loops all distributed loops can run asynchronously (see Section 3.4). If any of the tests failed, the pipeline loop will be treated as a serial loop.

3.3 Pipeline Setup

OpenMP provides directives (e.g. "`!$OMP FLUSH`") and library functions to perform the point-to-point synchronization, which makes the implementation of pipeline parallelism possible with directives. A potential pipeline loop (as introduced in Section 3) can be identified by checking the dependence vectors, which are created from the symbolic information. In order to set up a pipeline, an outside loop nesting is required.

To illustrate the concept, let us look at an example in Figure 3. Loop S2 is a potential pipeline loop which is nested inside loop S1. Three small code sections are inserted (as marked with bold font in the figure). The first section before loop S1 sets up a shared array (`ISYNC`) for the point-to-point synchronization. The second section right above loop S2 synchronizes the current thread (`IAM`) with the previous thread (`IAM-1`). Functionally it is similar to the "receive" in the message passing, which waits for the completion of the previous thread in action. The last section after loop S2 provides the "send" functionality to inform that the result from the current thread is available for the next thread by updating `ISYNC`. The two "`DO WHILE`" loops are used

to wait for neighboring threads to finish their work. The "NOWAIT" clause at the end of loop S2 is to avoid the implicit synchronization at the end of a parallel DO loop.

Setting up a pipeline across subroutine boundaries is described in Section 3.2. The pipeline algorithm is used when parallelizing the NAS benchmark LU (see Section 4.1.2).

```
S1   do j = 2, ny
S2     do i = 2, nx
S3       a(i,j) = a(i,j)
   &       + a(i-1,j)
   &       + a(i,j-1)
       end do
     end do
```

```
!$OMP PARALLEL PRIVATE(I,J,IAM,NEIGH)
      IAM = OMP_GET_THREAD_NUM()
      ISYNC(IAM) = 0
!$OMP BARRIER
  S1  DO J = 2, NY
        IF (IAM .GT. 0) THEN
          NEIGH = IAM - 1
          DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP       FLUSH(ISYNC)
          END DO
          ISYNC(NEIGH) = 0
!$OMP     FLUSH(ISYNC)
        ENDIF
!$OMP   DO
  S2    DO I = 2, NX
  S3      A(I,J) = A(I,J)
     &       + A(I-1,J) + A(I,J-1)
        END DO
!$OMP   END DO NOWAIT
        IF (IAM .LT. MAXTHREADNUM) THEN
          DO WHILE (ISYNC(IAM) .EQ. 1)
!$OMP       FLUSH(ISYNC)
          END DO
          ISYNC(IAM) = 1
!$OMP     FLUSH(ISYNC)
        ENDIF
      END DO
!$OMP END PARALLEL
```

Figure 3: Example of the pipeline implementation with point-to-point synchronization directives in OpenMP.

3.4 End-of-loop Synchronization

Synchronization is used to ensure the correctness of program execution after a parallel construct (such as END PARALLEL or END DO). By default, synchronization is added at the end of a parallel loop. Sometime the synchronization at the end of a loop can be eliminated to reduce the overhead. For example, in the code given in Figure 4, a "*NOWAIT*" clause can be used with the ENDDO directive to avoid many synchronizations inside the J loop.

To be able to execute two loops asynchronously and to avoid a thread synchronization directive between them we have to perform a number of tests aside from the dependence information provided by CAPTools. The tests verify whether a thread executing a portion of the instructions of one loop will not read/write data read/written by a different thread executing a portion of another loop. Hence, for each non-private array we check that the set of written locations of the array by the first thread and the set of read/written locations of the array by the second thread do not intersect. The condition is known as the Bernstein condition (see [12]). If Bernstein condition (BC) is true the loops can be executed asynchronously.

```
!$OMP PARALLEL PRIVATE(J,I)
      DO J=2, NY-1
!$OMP   DO
        DO I=1,NX
          A(I,J) = A(I,J) + A(I,J-1)
        END DO
!$OMP   END DO NOWAIT
      END DO
!$OMP END PARALLEL
```

Figure 4. Code with removable synchronization at the end of the loop

The BC test is performed in two steps. The final decision is made conservatively: if there is no proof that BC is true it set to be false. We assume that the same number of threads execute both loops, the number of threads is larger than 1 and there is an array read/written in both loops.

Check the number of loop iterations. Since the number of the threads can be arbitrary, the number of iterations performed in each loop must be the same. If it cannot be proved that the number of iterations is the same for both loops the Bernstein condition set to be false.

Compare array indices. For each reference to a non-privatizable array in LHS of one loop and for each reference to the same array in another loop we compare the array indices. If we can not prove for at least one dimension that indices in both references are different then we set BC to be false. The condition can be relaxed if we assume the same thread schedule is used for both loops.

Let the array indices be represented as functions of loop indices (*L1* and *L2*), so we want to prove that

$$index1(L1) \mathrel{!}= index2(L2), \text{ with } L1=1,...,d;\ L2=1,...,d \text{ and constraint } \lfloor L1/w \rfloor = \lfloor L2/w \rfloor \quad (1)$$

where d is the number of iterations and $w=\lceil d/number\_of\_threads \rceil$ is the workload per thread. The constraint means that the indices are computed by different threads. Now we represent *index1(L1)* and *index2(L2)* as symbolic polynomials of *L1* and *L2*. If the polynomials have the same coefficients then we set BC to be true. Otherwise, if degree of at least one polynomial is larger than 1 we set BC to be false since it would be too difficult to prove (1).

For linear index functions, we have to show that the system of equations

$$a1*L1 - a2*L2 = f, \text{ with } L1=1,...,d;\ L2=1,...,d \text{ and constraint } \lfloor L1/w \rfloor = \lfloor L2/w \rfloor \quad (2)$$

has no solutions. We apply Euclidean algorithm to find general solution of (2) in the form

$L1 = L10+a2*e$, $L2 = L20+a1*e$, for a set of integer e: $el<=e<=eh$. (3)

After that, test of the Bernstein condition is straightforward.

3.5 Variable Usage Analysis

Properly identifying variable usage is very important for the parallel performance and the correctness of program execution. Variables that would cause memory access conflict among threads need to be privatized so that each thread will work on a local copy. For cases where the privatization is not possible, for instance, a variable would partially be updated by each thread, the variable should be treated as shared and the work in the loop or region can only be executed in sequential (except for the reduction operation). With OpenMP, if a private variable needs its initial value from outside a parallel region, the `FIRSTPRIVATE` clause can be used to obtain an initial copy of the original value; if a private variable is used after a parallel region, the `LASTPRIVATE` clause can be used to update the shared variable.

Private variables are identified by examining the data dependence information, in particular, output dependence for memory access conflict and true dependence for value assignment. Partial updating of variables is checked by examining array index expressions.

The reduction operation is commonly encountered in calculation. A typical implementation of parallel reduction has a private copy of each reduction that is first created for each thread, the local value is calculated on each thread, and the global copy is updated according to the reduction operator. OpenMP only supports reductions for scalar values. For array, we first transform the code section to create a local array and, then, update the global copy in a `CRITICAL` section.

3.6 Private Variables in Common Blocks

For a private variable, each thread keeps a local copy and the original shared variable is untouched during the course of updating the local copy. If a variable declared in a common block is private in a loop, changes made to the variable through a subroutine call may not be updated properly for the local copy of this variable. If all the variables in the common block are privatizable in the whole program, the common block can be declared as *threadprivate*. However, if the common block can not be thread-privatized, additional care is needed to treat the private variable. For example, in Figure 5 the private array B is assigned inside subroutine `SUB` via the common block `/CSUB/` in loop S2. In order to have change made in `SUB` available to the local copy of B inside loop S2, the variable B needs to be passed through arguments of subroutine `SUB`.

```
S1  common /csub/ b(100),
    &           a(100,100)
 S2  do j=1, ny
 S3    call sub(j, nx)
       do i=1, nx
         a(i,j) = b(i)
       end do
     end do

 S4  subroutine sub(j, nx)
 S5  common /csub/ c(100),
    &           a(100,100)
     c(1) = a(1,j)
     c(nx) = a(nx,j)
     do i=2, nx-1
       c(i) = (a(i+1,j) +
    &          a(i-1,j))*0.5
     end do
     return
     end
```

```
  S1  COMMON/CSUB/B(100),A(100,100)
!$OMP PARALLEL DO PRIVATE(I,J,B)
  S2  DO J=1, NY
  S3    CALL SUB(J, NX, B)
        DO I=1, NX
          A(I,J) = B(I)
        END DO
      END DO
!$OMP END PARALLEL DO

  S4  SUBROUTINE SUB(J, NX, C)
  S5  COMMON/CSUB/C_CAP(100),A(100,100)
  S6  DIMENSION C(100)
      C(1) = A(1,J)
      C(NX) = A(NX,J)
      DO I=2, NX-1
        C(I) = (A(I+1,J)+A(I-1,J))*0.5
      END DO
      RETURN
      END
```

Figure 5: Example of treating a private variable in a common block. The private variable B is added to the argument list and the corresponding variable C used in the common block in SUB is renamed to C_CAP to avoid usage conflict.

The following algorithm is implemented to treat private variables in common blocks. The algorithm identifies and performs the necessary code transformation as described in the above example. Let us use the following convention: R_INSIDE for routine called inside a distributed loop, R_OUTSIDE for routine called outside a distributed loop, R_CALL for routine in a call statement, R_CALLBY for routine that makes a call to subroutine, and V (or VC, VD, VN) for a variable named in a routine. The algorithm starts with private variables listed for a parallel region in routine R_ORIG, one variable at a time. Each call statement in the parallel region is checked. A list of routine-variable pairs (R_CALL,VC) is stored in RenList during the process to track where private variables appear in common blocks and. These variables in common blocks are renamed at the end.

```
TreatPrivate(V, R_ORIG, callstatement) {
    check V usage in callstatement
    if V is not used in the call (via dependences) || is on the command parse tree
            || is not defined in a regular common block in a subroutine along the call path
        return
    TreatVinCall (VC, R_CALL) (V is referred as VC in R_CALL) {
        if VC is in the argument list of R_CALL
            return VC
        if R_CALL is R_OUTSIDE {
```

```
            if VC is not declared in R_CALL {
                replicate the common block in which V is named as VN
                set VC to VN from the common block
                set V to VN in the private variable list if R_CALL==R_ORIG
            }
        }
        else {
            add VC to the argument list of R_CALL
            if VC is defined in a common block of R_CALL
                add R_CALL & VC to RenList (for variable renaming later on)
            else declare VC in R_CALL
            for each calledby statement of R_CALL {
                VD is the name of VC used in R_CALLBY
                TreatVinCall(VD, R_CALLBY) and set VD to the returned value
                add VD to the argument of call statement to R_CALL in R_CALLBY
            }
            for each call statement in R_CALL
                TreatPrivate(VC, R_CALL, callstatement_in_R_CALL)
        }
        return VC
    }
}
```

Rename common block variables listed in RenList.

This concludes our treatment of private variables in common blocks.

3.7 Routine Duplication

Routine duplication is performed after all the analyses are done but before directives are inserted. A routine needs to be duplicated if it is called both inside and outside of a distributed loop and the routine itself contains distributed loops. Routine duplication ensures that no directive is inserted in the routine called inside a distributed loop.

An example of routine duplication is illustrated in Figure 6. Routine "init" is called before and inside loop S2. It is duplicated (as CAP_INIT) so that directives are inserted for the parallel loop (S5) inside INIT and the version without directives (CAP_INIT) can be called inside S2 (rename INIT to CAP_INIT at S3).

```
S1   call init(nx,1,a)
S2   do j = 2, ny
S3     call init(nx,j,a)
     end do
     ...

S4   subroutine init(nx,j,a)
     real a(100,100)
S5   do i = 1, nx
       a(i,j) = ...
     end do
     end
```

```
  S1   CALL INIT(NX,1,A)
!$OMP PARALLEL DO PRIVATE(J)
  S2   DO J = 2, NY
  S3     CALL CAP_INIT(NX,J,A)
       END DO
!$OMP END PARALLEL DO

  S4   SUBROUTINE INIT(NX,J,A)
       REAL A(100,100)
!$OMP PARALLEL DO PRIVATE(I)
  S5   DO I = 1, NX
         A(I,J) = ...
       END DO
!$OMP END PARALLEL DO
       END
  S6   SUBROUTINE CAP_INIT(NX,J,A)
       REAL A(100,100)
       DO I = 1, NX
         A(I,J) = ...
       END DO
       END
```

Figure 6: Example of routine duplication.

4 Case Studies

Following the procedure described in Section 2.3 we have used CAPO to parallelize NAS parallel benchmarks and ARC3D. First, each sequential code was analyzed for interprocedural data dependences. This step was the most computationally-intensive part of the process. The result was saved to an application database for later use. The loop and region level analysis was then carried out to produce OpenMP directives. At this point, the user inspects the result and decides if any changes are needed. Users assist the analysis by, for example, providing additional information on input parameters and removing any false dependences that could not be resolved by the tool. This is an iterative process, with user interaction involved.

In the case studies, we used an SGI workstation (R5K, 150MHz) to run CAPO. The resulting OpenMP codes were tested on an SGI Origin2000 system, which consisted of 64 CPUs and 16 GB globally addressable memory. Each CPU in the system is a R10K 195 MHz processor with 32KB primary data cache and 4MB secondary data cache. The SGI's MIPSpro Fortran 77 compiler (7.2.1) was used for compilation with the "`-O3 -mp`" flag.

4.1 NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB's) were derived from computational fluid dynamics (CFD) codes. They were designed to compare the performance of parallel computers and are widely recognized as a standard indicator of computer performance. NPB consists of five kernels and three simulated CFD applications derived from important classes of aerophysics applications.

The five kernels mimic the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes. Details of the benchmark specifications can be found in [3] and the MPI implementations of NPB are described in [4].

In this study we used six benchmarks (LU, SP, BT, FT, MG and CG) from the sequential version of NPB2.3 [4] with additional optimization described in [9]. Parallelization of the benchmarks with CAPO is straightforward except for FT where additional user interaction was needed. In all cases, the parallelization process for each benchmark took from tens of minutes up to one hour, most of the time was spent in data dependence analysis. The performance of CAPO generated codes is summarized in Figure 7 together with comparison to other parallel versions of NPB: MPI from NPB2.3, hand-coded OpenMP [9], and versions generated with SGI-pfa [19].

4.1.1 BT and SP

Code structures of BT and SP are similar. User knowledge on the grid size was entered for the dependence analysis in which the majority of the analysis time was spent. CAPO was able to locate effective parallelization at the outer-most loop level. OpenMP directives were inserted automatically without any change. Figure 7 shows that the performance of CAPO-BT and SP is within 10% to the hand-coded OpenMP version and close to the MPI version. The "SGI-pfa" curves represent results from the parallel version generated by SGI-fpa without any change for SP and with user optimization for BT (see [19] for details). The lower performance illustrates the importance of accurate interprocedural dependence analysis that usually can not be emphasized in a compiler. It should be pointed out that the sequential version used in the SGI-pfa study was not optimized, thus, the sequential performance needs to be counted for the comparison.

4.1.2 LU

There are at least two ways to parallelize the SSOR algorithm in LU: *pipelining* and *hyperplane*. The pipeline method was adopted in the hand-coded MPI version. The study in [9] also indicated that a pipeline implementation performed better than a hyperplane implementation in LU. Following the dependence analysis and using the point-to-point synchronization provided by OpenMP directives and illustrated in Section 3.3, CAPO was able to automatically generate a parallel code with the pipeline algorithm. It has the same performance as the hand-coded OpenMP as indicated in the performance curve in Figure 7. However, the directive version does not scale as well as the MPI version. We attribute this performance degradation in the directive implementation to less data locality and larger synchronization overhead in the 1-D pipeline as compared to the 2-D pipeline used in the message passing version.

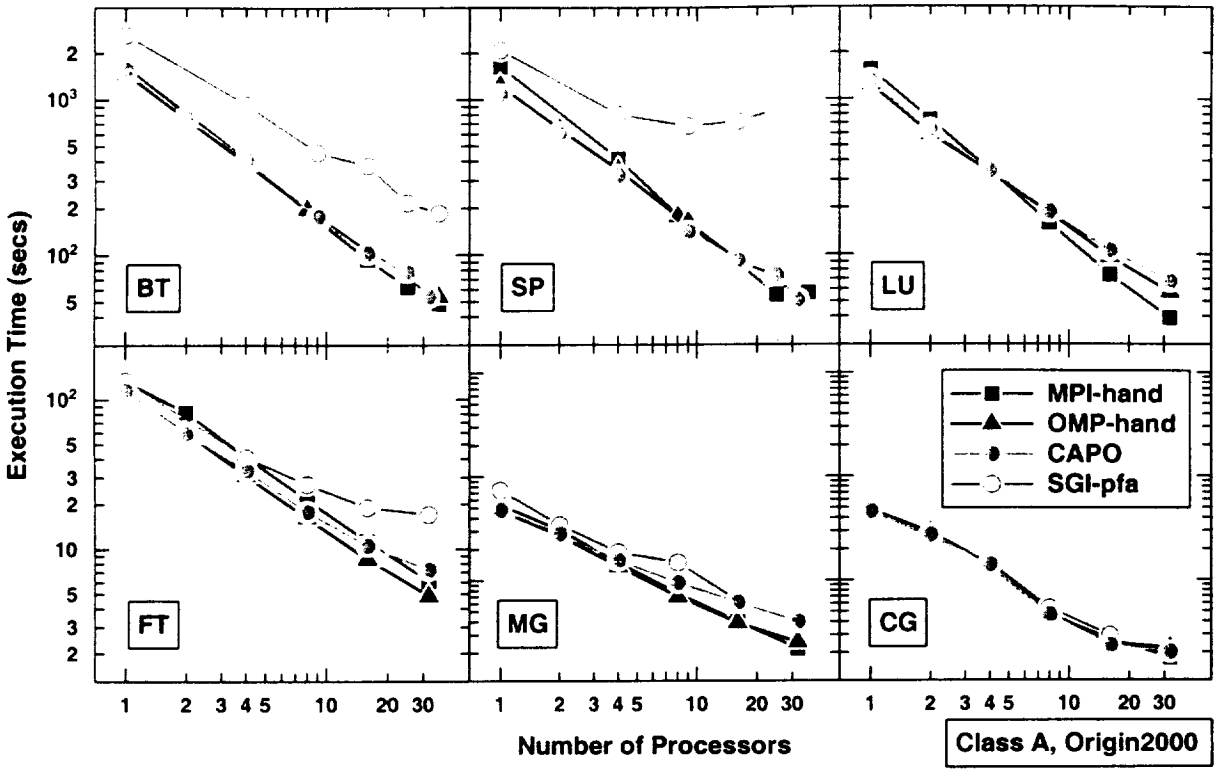

Figure 7: Comparison of the OpenMP NPB generated by CAPO with other parallel versions: MPI from NPB2.3, OpenMP by hand, and SGI-pfa.

4.1.3 FT

The basic loop structure for the Fast Fourier Transform (FFT) in one dimension in FT is as follows.

```
DO K = 1, D3
  DO J = 1, D2
    DO I = 1, D1
      Y(I) = X(I, J, K)
    END DO
    CALL CFFTZ(..., Y)
    DO I = 1, D1
      X(I, J, K) = Y(I)
    END DO
  END DO
END DO
```

A slice of the 3-D data (X) is first copied to a 1-D work array (Y). The 1-D FFT routine CFFTZ is called to work on Y. The returned result in Y is then copied back to the 3-D array (X). Due to the complicated pattern of loop limits inside CFFTZ, CAPTools could not disprove the loop-carried dependences by the working array Y for loop K. These dependences were deleted by the user in CAPO to identify the K loop as a parallel loop. The same treatment has been applied to all three directions of FFT.

The resulted parallel code gave a reasonable performance as indicated by the curve with filled circles in Figure 7. It does not scale as well as the hand-coded versions (both in MPI and OpenMP), mainly due to the unparallelized code section for matrix creation. The creation of matrix elements in FT was artificially done with random number generators, which could not be parallelized by CAPO without restructuring the code section as done in the hand-coded version. The better performance of the hand-coded directive version in compared with the MPI version was due to the elimination of a 3-D data array which was needed in the MPI version. Again, SGI-fpa generated code with lower performance.

4.1.4 CG and MG

All the parallel versions achieved similar results for CG and MG close results, especially CG. The directive code generated by CAPO for MG performs 36% worse on 32 processors than the hand-coded version, primarily due to an unparallelized loop in routine `norm2u3`. The loop contains several reduction operations of different types, which were not detected by CAPO, thus, the routine was ran in serial. Although this routine takes only about 2% of the total execution time on a single node, it translates into a large portion of the parallel execution on large number of processors, for example, 40% on 32 processors.

4.2 ARC3D

ARC3D is a moderate-size computational fluid dynamics (CFD) application [17]. It solves Euler and Navier-Stokes equations in three dimensions using a single rectilinear grid. Unlike the NAS parallel benchmarks, ARC3D contains turbulent models and more realistic boundary conditions. The Beam-Warming algorithm is used to approximately factorize an implicit scheme of finite difference equations, which is then solved in three directions. The solver sweeps through each of the cardinal directions one at a time, with partial updating of the fields after each sweep.

A parallel version of ARC3D with SGI multi-tasking directives was implemented by hand for the SGI Origin2000 system [18]. Much of the work in this effort involved the optimization of cache performance and array locality on the distributed shared-memory system. Preliminary results show promise for achieving very high sustained performance levels. Results of parallelizing ARC3D with CAPTools for distributed memory system in the message passing paradigm were reported in [11].

For generating the OpenMP parallel version of ARC3D, we used the same serial code as used in the hand parallelization. The serial version was optimized for cache performance. CAPO was used to perform data dependence analysis, loop and region analysis, and OpenMP directives insertion. The generated parallel version was unchanged and tested on the Origin2000. The result for a 194x194x194-size problem is shown in Figure 8 along with the version parallelized by hand. The result from the message-passing version generated by CAPTools from the same serial version is also included in the figure.

As one can see from the figure, the OpenMP version generated by CAPO is essentially the same as the hand-coded version in performance. This is indicative of the accurate data dependence analysis from CAPTools and sufficient parallelism that was explored in the outer-most loop level. The MPI version is about 10% worse than the directive-based versions. The MPI version uses extra buffers for communication and this could contribute to the increase of execution time. It should be pointed out, however, that the results did not show the effort in cache optimization of the serial code, which would help the parallel performance as well.

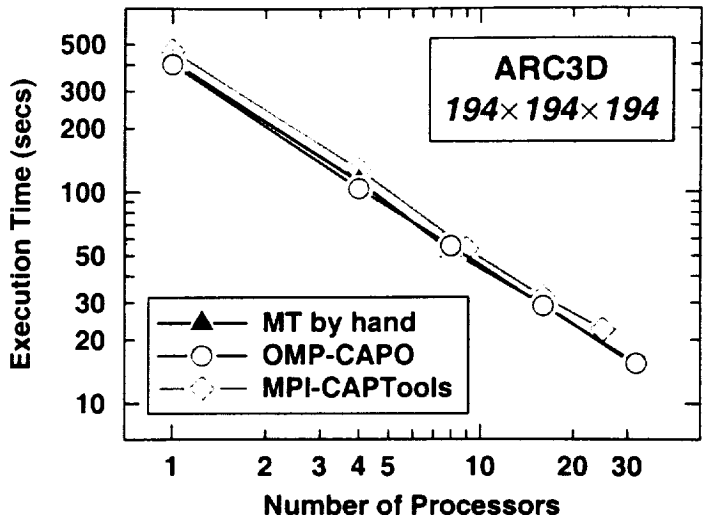

Figure 8: Comparison of execution times of two parallel versions of ARC3D: the OpenMP directive version generated by CAPTools and a directive version created by hand.

5 Related Work

There are a number of tools developed for code parallelization on both distributed and shared memory systems. The KAPro-toolkit [13] from Kuck and Associates, Inc. performs data dependence analysis and automatically inserts OpenMP directives in a certain degree. KAI has also developed several useful tools to ensure the correctness of directives insertion and help user to profile parallel codes. The SUIF compilation system [20] from Standard is a research product that is targeted at parallel code optimization for shared-memory system at the compiler level.

The SGI's MIPSpro compiler includes a tool, PFA, that tries to automatically detect loop-level parallelism, insert compiler directives and transform loops to enhance their performance. SGI-PFA is available on the Origin2000. Due to the constraints on compilation time, the tool usually cannot perform a comprehensive dependence analysis, thus, the performance of generated parallel programs is very limited. User intervention with directives is usually necessary for better performance. For this purpose, Parallel Analyzer View (PAV), which annotate the results of dependence analysis of PFA and present them graphically, can be used to help user insert directives manually. More details of a study with SGI-PFA can be found in [19].

VAST/Parallel [16] from Pacific-Sierra Research is an automatic parallelizing preprocessor. The tool performs data dependence analysis for loop nests and supports the generation of OpenMP directives.

Parallelization tools like FORGExplorer [2] and CAPTools [7] emphasize the generation of message passing parallel codes for distributed memory systems. These tools can easily be extended to handle parallel codes in the shared-memory arena. Our work is such an example. As discussed in previous sections, the key to the success of our tool is the ability to obtain accurate data dependences combined with user guidance. An ability to handle large applications is also important.

6 Conclusion and Future Work

In summary, we have developed the tool CAPO that automatically generates directive-based parallel programs for shared memory machines. By taking advantage of the intensive data dependence analysis from CAPTools, CAPO has been able to produce parallel programs with performance close to hand-coded versions for NAS parallel benchmarks and ARC3D, as summarized in Table 1. This approach is different from parallel compilers in that it spends much of its time on whole program analysis to discover accurate dependence information. The generated parallel code is produced using a source-to-source transformation with very little modification to the original code and, therefore, is easily maintainable.

Table 1: Summary of CAPO applied on NAS Parallel Benchmarks and ARC3D. "Parallelization" includes the time spent in data dependence analysis.

Application	***NPB***		***ARC3D***
	BT,SP,LU	***FT,CG,MG***	
Code Size	~3000 lines/benchmark	~2000 lines/benchmark	~4000 lines
Parallelization	30 mins to 1 hour	10 mins to 30 mins	40 mins
Testing	1 day	1 day	1 day
Performance compared to hand-coded version	within 5-10%	within 10% for CG 30-36% for FT,MG	within 6%

Even though the tested problems are small to medium size, it does indicate that CAPO can generate very efficient parallel codes in a short period of time. However, for larger, more complex applications, it is our experience that the tool will not be able to generate efficient parallel codes without any user interactions. In fact, CAPO has been applied to parallelize large CFD applications (20K-100K lines) and the tool was able to point out a small percentage of code sections where user interactions were required. The result has been reported elsewhere [10].

Future work will be focused in the following areas:

- A performance model for optimal placement of directives.

- Apply data distribution directives (such as those defined by SGI) rather than relying on the automatic data placement policy, *First-Touch*, by the operating system to improve data layout and minimize number of costly remote memory reference.
- Develop a methodology to work in a hybrid approach to handle parallel applications in a heterogeneous environment or a cluster of SMP's. Exploring multi-level parallelism is important.
- Develop an integrated working environment for sequential optimization, code transformation, code parallelization, and performance analysis.

CAPO is available for testing. A copy of the tool can be obtained from one of the authors.

Acknowledgement: The authors wish to thank members of the CAPTools team at the University of Greenwich for their support on CAPTools and Dr. James Taft for providing the ARC3D source code used in the study. This work is supported by NASA Contract No. NAS2-14303 with MRJ Technology Solutions.

References

[1] J.R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," *ACM Trans. Programming Languages Systems*, 9 (1987) 491-542.

[2] Applied Parallel Research Inc., "FORGE Explorer," http://www.apri.com/.

[3] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The NAS Parallel Benchmarks," *NAS Technical Report RNR-91-002*, NASA Ames Research Center, Moffett Field, CA, 1991.

[4] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *RNR-95-020*, NASA Ames Research Center, 1995. NPB2.3, http://www.nas.nasa.gov/Software/NPB/.

[5] U. Banerjee, "Speedup of Ordinary Programs," *Ph.D Thesis*, University of Illinois at Urbana Champaign, 1979.

[6] High Performance Fortran Forum, "High Performance Fortran Language Specification," CRPC-TR92225, January 1997, http://www.crpc.rice.edu/CRPC/softlib/TRs_online.html.

[7] C.S. Ierotheou, S.P. Johnson, M. Cross, and P. Legget, "Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, 22 (1996) 163-195. (http://captools.gre.ac.uk/)

[8] S.P. Johnson, M. Cross, and M.G. Everett, "Exploitation of symbolic information in interprocedural dependence analysis," *Parallel Computing*, 22 (1996) 197-226.

[9] H. Jin, M. Frumkin and J. Yan., "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," *NAS Technical Report*, NAS-99-011, NASA Ames Research Center, 1999.

[10] H. Jin, M. Frumkin and J. Yan., "Use Computer-Aided Tools to Parallelize Large CFD Applications," to be presented at the CAS 2000 Workshop, NASA Ames Research Center.

[11] H. Jin, M. Hribar and J. Yan, "Parallelization of ARC3D with Computer-Aided Tools," *NAS Technical Report*, NAS-98-005, NASA Ames Research Center, 1998.

[12] C.H. Koelbel, D.B. Loverman, R. Shreiber, GL. Steele Jr., M.E. Zosel. The High Performance Fortran Handbook, MIT Press, 1994, page 193.

[13] Kuck and Associates, Inc., "Parallel Performance of Standard Codes on the Compaq Professional Workstation 8000: Experiences with Visual KAP and the KAP/Pro Toolset under Windows NT," Champaign, IL; "Assure/Guide Reference Manual," 1997.

[14] Amy W. Lim and Monica S. Lam. "Maximizing Parallelism and Minimizing Synchronization with Affine Transforms," The 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, Jan,. 1997.

[15] OpenMP Fortran/C Application Program Interface, http://www.openmp.org/.

[16] Pacific-Sierra Research, "VAST/Parallel Automatic Parallelizer," http://www.psrv.com/.

[17] T. H. Pulliam, "Solution Methods In Computational Fluid Dynamics," *Notes for the von K'arm'an Institute For Fluid Dynamics Lecture Series*, Rhode-St-Genese, Belgium, 1986.

[18] J. Taft, "Initial SGI Origin2000 Tests Show Promise for CFD Codes," *NAS News*, July-August, page 1, 1997. (http://www.nas.nasa.gov/Pubs/NASnews/97/07/article01.html)

[19] A. Waheed and J. Yan, "Parallelization of NAS Benchmarks for Shared Memory Multiprocessors," in Proceedings of High Performance Computing and Networking (HPCN Europe '98), Amsterdam, The Netherlands, April 21-23, 1998.

[20] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W.K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica Lam, and John Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," Computer Systems Laboratory, Stanford University, Stanford, CA.